NATIONAL UNIVERSITY OF SINGAPORE

SCHOOL OF COMPUTING
FINAL ASSESSMENT FOR
Semester 1 AY2018/2019

CS1010 Programming Methodology

November 2018 Time Allowed 2 Hours

INSTRUCTIONS TO CANDIDATES

1. This assessment paper contains 15 questions and comprises 14 printed pages, including this page.
2. Write all your answers in the answer sheet provided.
3. The total marks for this assessment is 70. Answer **ALL** questions.
4. This is an **OPEN BOOK** assessment.
5. You can assume that in all the code given, no overflow nor underflow will occur during execution. In addition, you can assume that all given inputs are valid and fits within the specified variable type.
6. You can assume all the necessary headers (`math.h`, `stdbool.h`, `cs1010.h`, etc.) are included. For brevity, the include directives are not shown. Further, not all variable declarations are shown. You can assume that all variables are properly declared with the right type.
7. State any additional assumption that you make.
8. Please write your student number only. Do not write your name.

Part I
Multiple Choice Questions (36 points)

For each of the questions below, **write your answer in the corresponding answer box on the answer sheet.** Each question is worth 3 points.

If multiple answers are equally appropriate, pick one and write the chosen answer in the answer box. Do NOT write more than one answer in the answer box.

1. (3 points) Consider the following three functions that take in a `long` value as input. Which of these three functions are equivalent? Two functions are equivalent if both return the same value when given the same input.

```
bool f(long x) {
  if (x % 10 == 0) {
    if (x < 0) {
      return true;
    }
  } else {
    if (x < 10) {
      return true;
    }
  }
  return false;
}

bool g(long x) {
  if (x > 10 && x % 10 > 0) {
    return false;
  }
  return true;
}

bool h(long x) {
  return (x < 0 && x % 10 == 0) || (x < 10);
}
```

A. `f` and `g` only

B. `g` and `h` only

C. `g` and `h` only

D. `f`, `g`, and `h`

Write `X` in the answer box if no two functions are equivalent.

Solution: This question assesses if you understand `if-else` statements and logical expressions.

X. None of the functions are equivalent. f(0) is false, but g(0) and h(0) are true. g(10) is true, but f(10) and h(10) are true.

Some students incorrectly thinks that `f` and `h` are equivalent. The function `h` is actually equivalent to:

```
bool f(long x) {
  if (x % 10 == 0) {
    if (x < 0) {
      return true;
    }
  }
  if (x < 10) {
    return true;
  }
  return false;
}
```

One of B or C should be `f` and `h` only. I am sorry for the typo but luckily this does not affect the answer. In fact, I think this typo saves some students who think that `f` and `h` are equivalent from choosing the wrong answer :) For this reason as well, I decided not to issue a correction during the exam despite many questions from you.

2. (3 points) Consider the following three functions that takes in an array with at least one element as input. Which of these three functions are equivalent? Two functions are equivalent if they always return the same value when given the same input.

```
long foo(long len, long a[len]) {
  for (long i = 0; i < len; i += 1) {
    if (a[i] % 2 == 0) {
      return i;
    }
  }
  return len;
}

long bar(long len, long a[len]) {
  long i = 0;
  bool done = false;
  while (!done) {
    if (a[i] % 2 == 0 || i >= len) {
      done = true;
    }
    i += 1;
  }
  return i;
}

long qux(long len, long a[len]) {
  long i = -1;
  do {
    i += 1;
  } while (a[i] % 2 != 0 && i < len);
  return i;
}
```

A. `foo` and `bar` only

B. `foo` and `qux` only

C. `bar` and `qux` only

D. `foo`, `bar`, and `qux`

Write X in the answer box if no two functions are equivalent.

Solution: This question assesses if you understand how arrays and the three looping structures in C behaves.

In the given functions, `bar` always returns `i` that is larger than the other two (due to `i += 1;` even after we found the even element in array a.

To work as intended (return the index of the first even array element, the code should look like:

```
long bar(long len, long a[len]) {
  long i = 0;
  bool done = false;
  while (!done) {
    if (a[i] % 2 == 0 || i >= len) {
      done = true;
    } else {
      i += 1;
    }
  }
  return i;
}
```

or,

```
long bar(long len, long a[len]) {
  long i = 0;
  while (true) {
    if (a[i] % 2 == 0 || i >= len) {
      break;
    }
  }
  return i;
}
```

In qux, since the loop condition checks `a[i] % 2 != 0` first before `i < len`, the code will access `a[len]`, which could be illegal if the input array is of length `len`. We cannot predict what would happen if such illegal read occurs (the C standard calls this *undefined behavior*).

In `foo`, no illegal access would occur as only array elements `a[0]` to `a[len-1]` are read.

The answer is X.

However, your experience with `clang` on Ubuntu is that, just reading the illegal elements would not crash the program, only leads to garbage values being read, and since we exit the loop anyway, it does not change the behavior of the program. As such, we also accept B (`foo` and qux are equivalent) as the answer.

3. (3 points) Suppose we allocate a dynamic array as follows:

```
char *str = calloc(len, sizeof(char));
```

Which of the following snippet would lead to illegal access of memory?

```
void moo(char *str, long len) {
  for (long i = 0; i <= len; i++) {
    str[i] = 'a';
  }
  cs1010_println_string(str);
}

void baa(char *str, long len) {
  for (long i = 0; i < len; i++) {
    str[i] = 'a';
  }
  cs1010_println_string(str);
}

void quack(char *str, long len) {
  for (long i = 1; i < len-1; i++) {
    str[i] = 'a';
  }
  cs1010_println_string(str);
}
```

A. `moo` and `baa` only

B. `moo` and `quack` only

C. `moo`, `baa`, and `quack`

D. `moo` only

E. `quack` only

Write `X` in the answer box if none of the combinations above is correct.

Solution: A.

`moo` would lead to an illegal write to `a[len]`. `baa` would lead to illegal read (in print) since the string is no longer null-terminated (a[len-1] is set to 'a'). `quack` would read an uninitialized char but it is not illegal memory access.

This question assesses if you are familiar with the convention of strings as well as the bounds of an array.

4. (3 points) Consider the code below:

```
double d;
while (d > 0) {
  d -= 1;
}
```

Which of the following statements about the code above is TRUE?

A. The code causes a compilation error because `-=` can only be used with integer types

B. The code causes a compilation error because `d` is not initialized

C. The code causes a compilation error because we should cast `0` and `1` to `double`

D. The code may lead to an infinite loop because `d` is not initialized

E. The code may lead to an infinite loop because `0` cannot be precisely represented by the type `double`

Write `X` in the answer box if none of the statements above is true.

Solution: X.

This question checks if you are familiar with the type `double` in C.

5. (3 points) Suppose we define a macro AVERAGE as follows:

```
#define AVERAGE(x,y) (0.5 * x) + (0.5 * y)
```

What is the value of `double_average`, round to the nearest integer, after executing the three lines of code below?

```
long x = 10;
long y = 20;
double double_average = 2*AVERAGE(x, y);
```

A. 20

B. 25

C. 30

D. 35

E. 40

Write `X` in the answer box if none of the answers above is correct.

Solution: A. The expression gets expended into `2*0.5*x + 0.5*y` = `x + 0.5*y` = 20.

This question checks if you are aware of how macro works in C.

6. (3 points) Consider the code below:

```
void tata(double *ptr, double x) {
  ptr = &x;
}

void titi(double *ptr, double *x) {
  *ptr = *x;
}

void tete(double **ptr, double x) {
  *ptr = &x;
}

void tutu(double **ptr, double *x) {
  *ptr = x;
}

int main()
{
  double *ptr;
  double x;
  double y;
  ptr = &y;
  // Line B
}
```

Which of the following function invocation at Line B would cause the assertion `ptr == &x` to be true?

A. `tata(ptr, x);`

B. `titi(ptr, &x);`

C. `tete(&ptr, x);`

D. `tutu(&ptr, &x);`

E. `tutu(*ptr, *x);`

Write `X` in the answer box if none of the answers above is correct.

Solution: D.

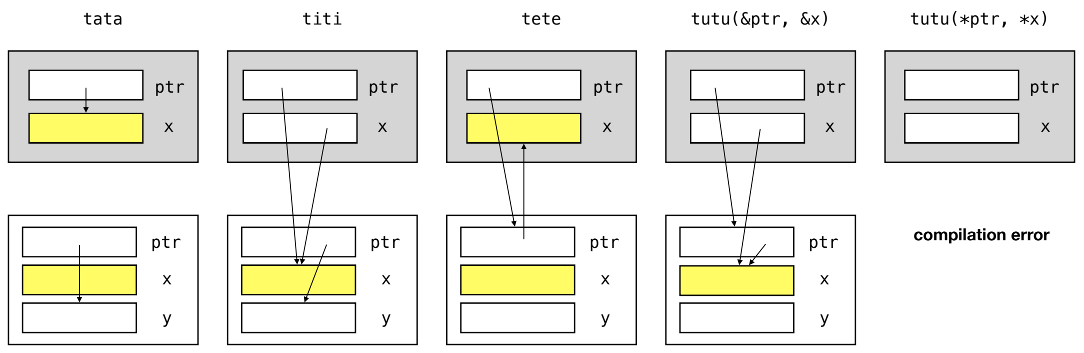
Here, we need to write a function that changes the ptr so that it points to the address of `x`. We must have access to the address of `x` within the function. This observation eliminates choices A and C, which pass the *value* of `x` into the function.

We also need to update `ptr` to point to a new location – hence we need to pass `ptr` by reference to the function. This eliminates A and B.

Only D, E, and X remain. A quick check should eliminate E since the type does not even match.

Let's check if `tutu` does what it supposed to do: `*ptr = x`. The `x` in `tutu` points to the `x` in `main`. The `ptr` in `tutu` points to the `ptr` in `main`. `*ptr = x` in `tutu` thus makes the `ptr` in `main` points to `x` in `main`.o

It is useful to draw the call stack to understand the intricacy of this. The following shows the different versions of the functions just before `tata/titi/tete/tutu` exits.



7. (3 points) Which of the following functions run in $O(n^2)$ time?

```
void egg(long n, long a[n]) {
  for (long i = 0; i < n/2; i += 2) {
    for (long j = i/2; j < n; j += 1) {
      cs1010_println_long(i);
      cs1010_println_long(j);
    }
  }
}

void ham(long n, long a[n]) {
  for (long i = 1; i < pow(2, n); i *= 2) {
    for (long j = 1; j < n; j += 1) {
      cs1010_println_long(i);
      cs1010_println_long(j);
    }
  }
}

void cheese(long n, long a[n]) {
  for (long i = 1; i < n*sqrt(n); i += 1) {
    for (long j = 1; j < sqrt(n); j += 1) {
      cs1010_println_long(i);
      cs1010_println_long(j);
    }
  }
}
```

A. `egg` only

B. `cheese` only

C. `ham` and `cheese` only

D. `egg` and `ham` only

E. `egg`, `ham`, and `cheese`

Write `X` in the answer box if none of the combinations above is correct.

Solution: The outer loop of `egg` loops O(n/4) time, while the inner loop loops O(n) time. Therefore, `egg` runs in $O(n^2)$ time.

The outer loop of `ham` loops O(n) time, while the inner loop also loops O(n) time. Therefore, `ham` also runs in $O(n^2)$ time.

The outer loop of `cheese` loops $O(n\sqrt{n})$ time, while the inner loop loops $O(\sqrt{n})$ time. Therefore, `ham` also runs in $O(n\sqrt{n} \times \sqrt{n}) = O(n^2)$ time.

The answer is E.

8. (3 points) Consider the code below:

```
bool mystery(long a[], long start, long end) {
  if (end > start) {
    return false;
  }
  long mid = (start + end)/2;
  long sum = 0;
  for (long i = start; i <= mid; i += 1) {
    sum += a[i];
  }
  if (sum == 0) {
    return true;
  }
  if (sum > 0) {
    return mystery(a, start, mid);
  }
  return mystery(a, mid+1, end);
}
```

What is the worst-case running time of `mystery`, expressed using Big-O notation, when the input is an array of size n?

A. $O(n^2)$

B. $O(n \log n)$

C. $O(n)$

D. $O(\log^2 n)$

E. $O(\log n)$

Write X in the answer box if none of the answers above is correct.

Solution: This question checks if you know how to analyze the running time of a recursive function. I botched this question badly due to the typo :(

This intended behavior of the function is half the array at every recursive call, and have a linear scan in each call. So the running time can be expressed as $T(n) = n + T(n/2)$, which, after expansion, becomes $T(n) = n + n/2 + n/4 + n/8 + ...1$ which is $O(n)$.

Our first error is the typo in the question (`end > start`), which would cause the function to be $O(1)$. Our second error is to correct it into (`end < start`), which could cause the function to recurse forever.

It should be `end <= start`.

We accept both C $O(n)$ and X (infinite loop) as the answer.

Questions 9 to 10 are based on the following function.

```
void do_something(long len, long a[len]) {
  long curr = 0;
  while (curr < len) {
    if (curr == 0 || a[curr] >= a[curr-1]) {
      curr += 1;
    } else {
      long temp = a[curr];
      a[curr] = a[curr - 1];
      a[curr - 1] = temp;
      curr -= 1;
      // Line A
    }
  }
}
```

9. (3 points) Suppose we have an array `long a[3] = {3, 1, 2};`. What is the content of the array a after calling `do_something(3, a);` ?

   A. `{3, 2, 1}`

   B. `{1, 2, 3}`

   C. `{3, 3, 3}`

   D. `{3, 1, 2}`

   E. `{2, 1, 3}`

   Write X in the answer box if none of the answers above is correct.

   Solution: B.

   This is a bit tedious but quite straightforward as you only need to trace through the code carefully step-by-step.

10. (3 points) Which of the following is a correct assertion at Line A of the `do_something`?

    A. `{ curr != 0 && a[curr-1] > a[curr-2] }`

    B. `{ curr != 0 && a[curr] > a[curr-1] }`

    C. `{ curr != 0 && a[curr + 1] > a[curr] }`

    D. `{ curr >= 0 && a[curr] > a[curr-1] }`

    E. `{ curr >= 0 && a[curr + 1] > a[curr] }`

    Write X in the answer box if none of the answers above is correct.

    Solution: E.

    Line A is in the `else` block. At the beginning of this block, we know that `curr == 0 || a[curr] >= a[curr-1]` is false. Applying De Morgan's law, we know what `curr != 0 && a[curr] < a[curr-1]` is true.

The next three lines swap `a[curr]` with `a[curr-1]`. After swapping, `curr != 0 && a[curr] > a[curr - 1]` must be true.

The final line of code before Line A is `curr -= 1`. Let's analyze the easy part first, we previously have `a[curr] > a[curr - 1]`, now that `curr` is one less than before, the following must be true: `a[curr + 1] > a[curr]`.

This eliminates A, B, and D.

What about `curr != 0`? After we decrement `curr` by 1, `curr` can now be 0 and so can be anything. Choice C is eliminated. We only need to verify that E is correct.

We need a stronger constraint on `curr`. Since we are accessing `a[curr]`, it is a hint that `curr` is no less than 0. (You should observe this pattern when you solve Question 9). A quick check on the code can confirm this: whenever `curr` reaches 0, the true block of the `if` statement increments `curr` so that `curr` is 1. So `curr >= 0` and E is correct.

Side note: this algorithm is a single loop $O(n^2)$ sorting algorithm called *stupid sort* or *gnome sort*.

11. (3 points) Trace through the code below:

```
#include "cs1010.h"

struct obj {
  long *ptr;
  long id;
};

void blah(struct obj *optr) {
  optr->id = 20;
  optr->ptr = &optr->id;
}

int main()
{
  struct obj o;
  long x = 10;

  o.ptr = &x;
  o.id = 0;

  blah(&o);

  cs1010_println_long(o.id);
  cs1010_println_long(*(o.ptr));

}
```

Which two numbers will be printed by the program above?

A. `20` and `10`

B. `0` and `10`

C. `0` and `0`

D. `10` and `10`

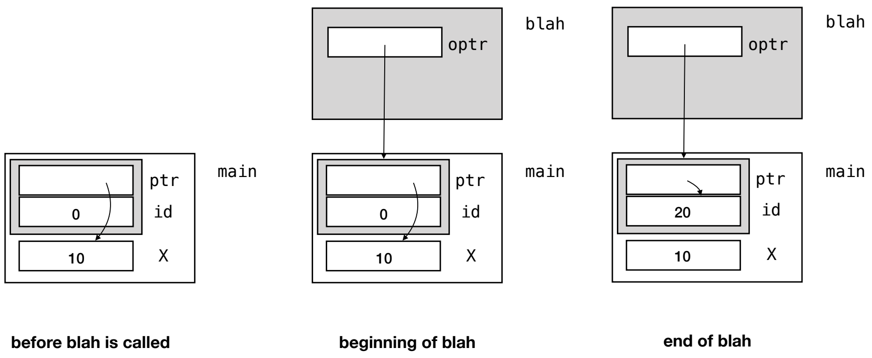
E. `20` and `20`

Write `X` in the answer box if none of the answers above is correct.

Solution: E.

Another pointer and call stack question – this time with `struct`.

As usual, it is useful to draw the call stack and trace through:



After calling `blah`, the pointer `o.ptr` points to `o.id`, so the print statements print out 20 and 20.

12. (3 points) Consider the following implementation of insertion sort:

```
void insert(long a[], long curr)
{
  long i = curr - 1;
  long temp = a[curr];
  while (temp <= a[i] && i >= 0) {
    a[i+1] = a[i];
    i -= 1;
  }
  a[i+1] = temp;
}

void insertion_sort(long n, long a[n]) {
  for (long curr = 1; curr < n; curr += 1) {
    insert(a, curr);
  }
}
```

Which of the following statement(s) is/are true about the code above:

(i) If the input array `a` contains `n` elements, all has the same value, `insertion_sort` takes $O(n^2)$ time.

(ii) If the input array `a` contains `n` distinct elements that are sorted in descending order, `insertion_sort` takes $O(n)$ time.

(iii) If the input array `a` contains `n` distinct elements that are sorted in ascending order, `insertion_sort` takes $O(1)$ time.

A. Only (i)

B. Only (i) and (ii)

C. Only (i) and (iii)

D. Only (ii) and (iii)

E. (i), (ii), and (iii)

Write `X` in the answer box if none of the combinations is correct.

Solution: A.

This is similar to the insertion sort algorithm given in the lecture notes, except that on Line 5, the continuing condition of the while loop is `temp <= a[i] && i >= 0` instead of `temp < a[i] && i >= 0`. Because of this, this implementation will continue to scan the array even if it finds a value that is strictly smaller than the value to-be-inserted.

Thus, (i) is true. Given an array with n elements of the same value, the algorithm still scans and re-insert every one of the elements

(ii) is false. An array that is inversely sorted is like the worst enemy of insertion sort. It will take $O(n^2)$.

(iii) is false. An array that is sorted is the best friend of insertion sort, but unfortunately, we still have to scan through the list to make sure that it is sorted. It will take $O(n)$ time.

Part II

Short Questions (34 points)

Answer all questions in the space provided on the answer sheet. Be succinct and write neatly.

13. (6 points) **Binary.** The following program `binary` generates all binary strings (i.e., strings consisting of `'0'` and `'1'` only) of a given length `n` recursively. For instance

```
ooiwt@pe101:~$ ./binary
2
00
01
10
11
ooiwt@pe101:~$ ./binary
3
000
001
010
011
100
101
110
111
```

The function `generate` below recursively generates all binary substrings of length `n - k` and prints out the binary string. The recursive calls, however, are missing.

Complete the function `generate`. Write only the missing lines on the answer sheet.

```
void generate(long n, char str[], long k) {
  if (k == n-1) {
    str[k] = '0';
    cs1010_println_string(str);
    str[k] = '1';
    cs1010_println_string(str);
    return;
  }

  // Missing Lines

}
```

Solution: The right answer is:

```
  str[k] = '0';
  generate(n, str, k+1);
  str[k] = '1';
  generate(n, str, k+1);
```

We do not deduct marks for syntax errors. If you get the four lines above in the right order, you should receive full marks.

A common bug is to do

```
generate(n, str, k+1);
str[k] = '0';
generate(n, str, k+1);
str[k] = '1';
```

The solution above gets 5 marks.

If you get only half of the lines, you get 3 marks at most. For instance:

```
str[k] = '0';
generate(n, str, k+1);
str[k] = '1';
```

Many students wrong something similar to this. We give 2 marks:

```
generate(n, str, k+1);
for (long i = k+1; i < n; i += 1) {
  str[i] = '0';
  generate(n, str, i+1);
  str[i] = '1';
}
```

Many students incorrectly print out the string in the answers, you will get 1 mark deducted.

Besides the above, there are many other variations of the solution, we try to give partial marks if possible. But the following solutions are among the popular ones that we did not give any marks:

```
generate(n, str, k+1);
```

or

```
// swap something
swap(..);
generate(n, str, k+1);
// swap back
swap(..);
```

Or some combinations of the above with `for` loops or `if` conditions.

14. (15 points) **Search.** Suppose we have the following function

```
long binsearch(const long list[], long i, long j, long q)
```

that looks for the item q in an array `list`, among items `list[i]` .. `list[j]` using binary search. The code for this function is omitted.

In an attempt to try to speed up binary search on a large array, Mario wrote the following function to find the starting point and end point in the array that contains q.

```
long narrowing_then_search(const long list[], long len, long q) {
  long start = 0;
  long end = 1;
  do {
    if (q == list[end]) {
      return end;
    }
    if (q < list[end]) {
      // Line F
      return binsearch(list, start, end, q);
    }
    start = end;
    end += 10; // Line G
    // Line H
  } while (end < len);
  // Line I
  return binsearch(list, start, len-1, q);
}
```

You can assume that the input array is already sorted, in non-decreasing order.

(a) (3 points) Write an assertion that relates q to `list[start]` in Line H.

(b) (3 points) Write an assertion that relates q to `list[start]` and `list[end]` in Line F.

(c) (3 points) Write an assertion that relates q to `list[start]` and `list[len-1]` in Line I.

(d) (3 points) What is the worst case running time, in Big-O notation, of this algorithm?

(e) (3 points) Suppose we change Line G to

`end *= 2;`

What is the worst-case running time, in Big-O notation, now?

Solution: The question does not assume that q must be somewhere between the list, although some students do. Due to the wordings of the questions, we will accept answers that assume that q is between `list[0]` and `list[len-1]`.

(a) `q > list[start]`

(b) `q < list[end]` or `list[start] <= q < list[end]`

(c) `q > list[start]` or `list[start] < q < list[len-1]`

(d) $O(n)$

(e) $O(\log n)$

Let me explain the solution using the more interesting case, where `q` is bounded by `list[0]` and `list[len-1]`. The idea behind Mario's algorithm is to chop the list into smaller ones, of size 10 each, then perform a binary search on the short list. (This turns out to be a bad idea, but, let's answer the questions first).

On Line H, we have "survive" the checks that `q == list[end]` and `q < list[end]`, so we know that `q > list[end]` and so `q > list[start]` just before Line G. Line G does not change `start` nor `q`, so that property `q > list[start]` still holds at Line H.

In the second iteration onwards, the property `q > list[start]` is true at Line F. And if we assume that `q >= list[0]`, we can ensure that `list[start] <= q < list[end]` at Line F. This property ensures that we can find `q` among `list[start..end]` with binary search.

Since we know `q > list[start]` holds at Line H, when we exit the loop, this property still holds. In addition, since we assume `q` is bounded by `list[len-1]`, we have `list[start] < q <= list[len-1]` This property ensures that we can again perform binary search for `q`.

Mario's intention is to speed up the search, but his algorithm actually runs in $O(n)$ time, since in the worse case, the algorithm needs to check $n/10$ times to find the right 10-element list to search for `q`. Binary search here runs in $O(1)$ time since the sub-array to binary-search in has at most 10 elements.

But, Mario's intention is not wrong. His algorithm can be fixed by changing Line G to `end *= 2`. The assertions above still remains, but now, the algorithm needs to check only $\log n$ times to find the right sub-array to search for `q`. In the worst case, the size of the sub-array is $n/2 = O(n)$. But the binary search on this is still $O(\log n)$, giving the total running time of $O(\log n)$.

Side note: After fixing Line G, the algorithm above becomes a search algorithm called *exponential search*. In practice, it outperforms binary search, especially when the item to search for is near the beginning of the array. Suppose the index of `q` is i. Then exponential search takes $O(\log i)$ time to find. Our traditional binary search always takes $O(\log n)$ independent of i.

15. (13 points) **Stack.**

Consider the code below:

```
void f(__ a, __ b) {
  *(b + 1) = a;
  b = &a;
  // Line D
}

int main() {
  long x[2] = {-5, 10};
  long *p;

  p = x;
  f(*p, x);

  // Line E
}
```

(a) (4 points) The types for the parameters a and b to function f is missing. Fill in the correct type of the parameters so that the compiler does not report any error or warning.

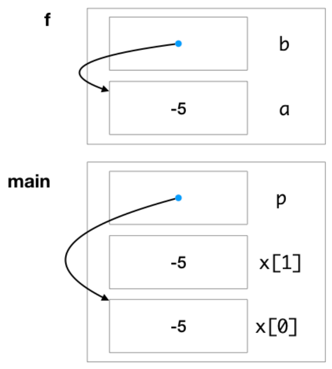
(b) (7 points) Draw the content of the call stack when the execution reaches Line D using the notations similar to what has been used in CS1010. Label all your call frames, variables, and values on the call stack. You may use arrows to denote pointers, instead of using the actual memory addresses.

(c) (2 points) What are the values in the array x after calling f, at Line E?

Solution:

(a) `void f(long a, long * b)`

Two marks for each type. This should be easy since you have read/written many programs that involves passing in arrays and pointers into a function.



(b)

Getting f, main, b, a, p, x[1], x[0] correct will give you one mark each. If you add extra stuff that does not belong to the stack, we deduct one mark each for each variable that does not belong to the stack.

We do not double penalize, so if you answer in (c) is wrong and is consistent with the content of `x` in (c), we do not penalize again in (b).

Some common mistakes include:

- Putting the array `x[]` in the heap or outside the call stack.
- Drawing `f`, `x` (in addition to the `x[0]` and `x[1]`), and `b+1` on the call stack.
- Treating `b` as an array (`b` is just a pointer).
- Storing `*p`, `*b`, etc on the stack. (Only the pointers are on the stack!)
- Drawing `main` and `f` out of order (`main` should be at the bottom). But we do not penalize for this.
- Drawing the call frame for `f` inside the call frame for `main`. This is usually done to languages with nested function but not to `C`.

(c) `x[0]` is -5, `x[1]` is -5.

END OF PAPER